Classes

Namespaces

Пространства имен нужны для логической группировки родственных типов.
Делают имя класса уникальным для компилятора.
Например, `System.Int32`, `System.Collections.Generic.List`.

`using` - директива заставляет компилятор добавлять этот префикс к классам, пока не найдет нужный класс.

В коде можно писать имя класса без namespace.

```
using System.IO;       // Классы для работы с файловой системой, потоками
using System.Collections;   // Все готовые коллекции
using System.Collections.Generic; // Обобщенные коллекции
using System.Linq;          // Набор хелперов для генерации LINQ запросов
using Newtonsoft.Json;      // Подключили сторонюю библиотеку
using Abbyy.Shared.Library; // Подключили свою отдельную библиотеку
...
var list = new List<int>();
```

Пространства имен и сборки могут не быть связаны друг с другом.

Типы одного пространства имен могут быть реализованы разными сборками.

Чтобы обезапасить от конфликтов имен рекомендуется использовать namespace, начинающийся с имени компании, потом название системы/подсистемы.

Если в двух namespace содержатся одинаковые классы, то:

- либо надо указывать полное имя класса с namespace
- либо можно, используя директиву `using`, задать alias для класса

```
using System.Windows.Forms;
using myButton = Abbyy.Shared.Controls.Button; // Добавляем alias для класса

...
var button = new myButton();
var button = new Abbyy.Shared.Controls.Button();
```

Members

Члены класса

```
using System;
internal class SomeType
{
    private class SomeNestedType { }    // Nested Type

    private int _x = 1;                 // Field
    private SomeNestedType Value;       // --

    internal SomeType(Int32 x) { }      // Конструкторы экземпляров

    internal protected void Method() {} // Method
    internal int Property { get;set;}   // Property
}
```

Access Modifiers

Модификаторы доступа определяют видимость элемента MSDN

- `public` - доступен в любых сборках
- `internal` - только в текущей сборке
- `private` - только в данном классе
- `protected` - в классе и его наследниках
- `internal protected` - в классах данной сборки или любых его наследниках (дурное название, конечно)
- `private protected` - доступен в классе и его наследниках в данной сборке

Можно определить сборку дружественной, чтобы internal можно было использовать в другой сборке.

- По-умолчанию, если не указать, будет private.
- Рекомендуется всегда явно указывать!
- Проверку доступа производит как базовый компилятор, так и JIT компилятор
- При наследовании от базового класса CLR позволяет снижать, но не повышать ограничения доступа к члену.

Насущный вопрос, если объявить класс `internal`, то с какой областью видимости объявлять методы/объекты в нём? `private` / `internal`.

- ! clr автоматически урежет права для экземплярных методов до уровня класса
  - исключения составляют `public` методы реализации интерфейсов или override методов публичного базового класса
- распространенная практика: задавать область видимость на уровне класса, а методы объявлять всегда `public`. Это дискуссионный момент, основной аргумент "за":
  - видимость для класса и его методов определяется в одном месте и это проще

```
internal static class Helper
{
    public static string Test() { return "Test"; }
}
```

Конструкторы

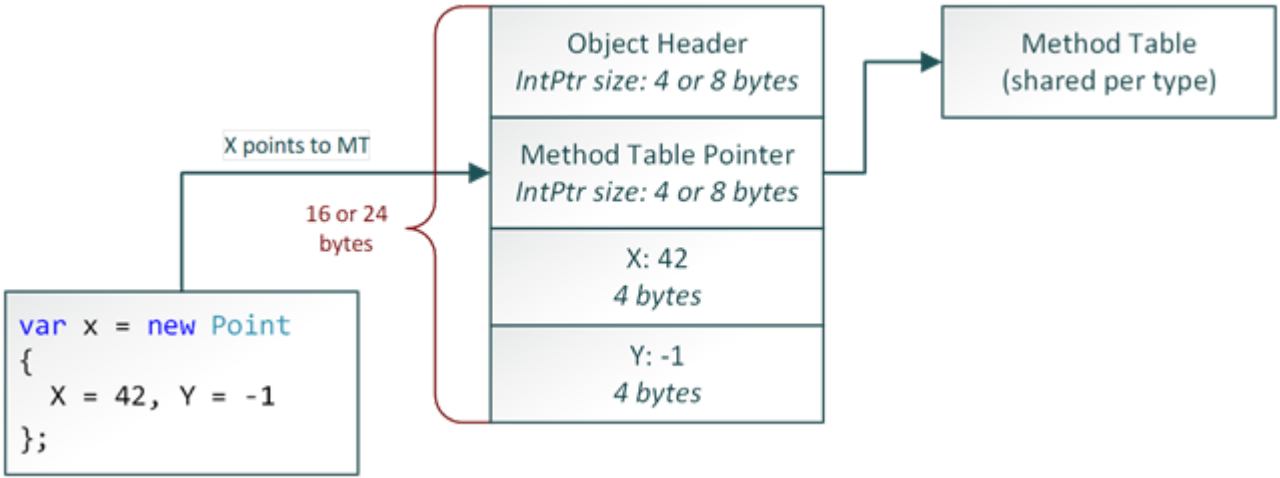
Все объекты создаются оператором new.

```
SomeType e = new SomeType(4);
SomeType e = new SomeType(3) { SomeProp = 5 }; // Инициализатор

e.SomeProp = 5  // Идентично
```

- Вычесляет количество необходимой памяти для экземпляра
- Выделяет память в куче, заполняя нулями
- Инициализирует указатель на объект
- Вызывает конструктор с параметрами
- Возвращает ссылку на созданный объект

У каждого managed объекта есть некоторый overhead:



- object header - used by clr
- “Why the managed object’s layout is so weird?”, is simple: “historical reasons” MSDN blog post

- Конструкторы позволяют инициализировать объект
- Если конструктор не указать, компилятор создаст пустой конструктор без параметров автоматически.
- Есть ключевое слово `this` для доступа к полям экземпляра

```
internal class SomeType
{
    private int _value;

    internal SomeType(int x)
    {
        _value = x;
        this._value = x; // Ключевое слово this
    }
}
var value = new SomeType(x);
```

- Конструкторы не наследуются

```
public class BaseType
{
    private readonly _x;
    public BaseType(int x)
    {
        _x = x;
    }
}

public class SomeType : BaseType
{
    public SomeType(int x) : base(x) { }
}
```

Еще пример с инициализатором:

```
internal class SomeType
{
    internal int Value;
}
var value = new SomeType(); // Инициализируем
value.Value = 10;
// Похожее поведение через инициализатор
var value = new SomeType { Value = 10 };
```

Если указать в классе поля со значениями, то компилятор по сути добавит инициализацию этого метода в цепочку вызова конструкторов:

```
internal class SomeType
{
    internal int Value = 10;
    public SomeType() {}
}
```

Methods

```
public class Example
{
    public int Add(int x, int y)
    {
        return x + y;
    }

    public void Print(int x)
    {
        Console.WriteLine(x);
    }

    // Expression-bodied method
    // без return
    // для методов/конструкторов/property содержащих одно выражение
    public void NewPrint(int x) => Console.WriteLine(x);
}
```

Необязательные параметры:

- Значения по умолчанию
  - идут после остальных параметров
  - определяются на этапе компиляции, поэтому либо примитивные типы, либо такие значения как `null`, `default`, `new(...)`
  - не задаются для `ref` / `out` параметров

```
public static int Method(int x, int y=0, int z=0,
    String s = "A", DateTime dt = default(DateTime), Guid guid = new Guid())
{
    return x + y + z;
}

int i = Method(5);
int u = Method(5, z:4); // Именованный параметр
```

Перегрузка методов:

```
public class Example
{
    public void Test(int a)
    {
        Console.WriteLine($"{a}");
    }
    public void Test(int a, int b)
    {
        Console.WriteLine($"{a} {b}");
    }
    public int Test(int a, int b, int c)
    {
        Console.WriteLine($"{a} {b} {c}");
        return a + b + c;
    }
    public void Test(double a, double b)
    {
        Console.WriteLine($"{a} {b}");
    }
```

- `ref` / `out` - заставляют компилятор передавать значение параметра по ссылке MSDN
- не вызывают boxing/unboxing, передается ссылка на стек
- оба типа компилируются в одинаковый IL код (отличается один бит в метаданных)
- `ref` - должен быть инициализирован до вызова метода

```
static void ExampleReference(ref int x, int y)
{
    x = x + y;
}
static void ExampleValue(int x, int y)
{
    x = x + y;
}

int x = 1;
int y = 2;
ExampleValue(x, y);
Console.WriteLine(x); // 1
ExampleReference(ref a, b);
Console.WriteLine(x); // 3
```

- `out` - позволяет не инициализировать параметр до вызова метода.
- метод не может читать значение параметра, должен сам инициализировать его обязательно
- указание `ref`/`out` при вызове метода сделано для более наглядного использования кода (плюс это важно т.к. могут быть перегрузки методов)

```
static try TryParseInt(string value, out int result)
{
   // реализация
}

int result;
bool isParsed = TryParseInt("33", out result);
```

Рассмотрим пример, когда передается ссылочный параметр:

```
class Product
{
    public int X { get; set; }
    public int Y { get; set; }
}
static void ChangeByReference(ref Product itemRef)
{
    itemRef = new Product { X = 1 };
    itemRef.Y = 2;
}

Product item = new Product { X = 4, Y = 5};
System.Console.WriteLine($"{item.X}, {item.Y}"); // 4, 5
ChangeByReference(ref item);
System.Console.WriteLine($"{item.X}, {item.Y}"); // 1, 2
```

Рекомендации по использованию `ref` / `out`:

- Лучше усложнить возвращаемый объект (или сделать DTO дополнительное), чем добавлять дополнительные ref / out параметры!
- Использование `ref / out` не к месту - признак плохой декомпозиции на методы и классы!
- Не используйте `ref` без крайней необходимости !!! Трижды подумайте и обойдитесь без него!
    - Он сильно ухудшает читаемость и поддерживаемость кода
    - Без него код становится проще и предсказуемее
- Используйте `out` со значимыми типами
- Если нужно внутри метода изменить состояние объекта, то возвращайте измененное состояние и присваивайте его экземпляру `value = methodThatChange(value)` - это явно показывает, что вы изменяете объект

Передача массива параметров, когда мы не знаем количество параметров:

- `params` может пометить только последний элемент
- только одномерный массив 😃 произвольного типа

```
public static int Add(params int[] values)
{
    int sum = 0;
    if (values != null);
    {
        for (int x = 0; x < values.Length; x++)
            sum += values[x];
    }
    return sum;
}
...
int result = Add(new int[] {1, 2, 3, 4, 5});
result = Add(1, 2, 3, 4, 5);
result = Add(); // Все варианты валидны
result = Add(null);
```

Local functions

Можно объявлять анонимные методы прямо в теле методов (подробнее потом при сравнении с делегатами):

- Локальные переменные передаются в метод через ref
- Не создает делегат (нет аллокации на него)
- Можно использовать итераторы `yield return`

```
public double Do(double a, double b, double c)
{
    var resultA = f(a);
    var resultB = f(b);
    return resultA + resultB;

    double f(double x) => 2 * x + 3 + c;
}
```

Deconstructors

Можно возвращать набор полей и делать ему "deconstructing":

Подробнее о `Tuple` в лекции generic.

```
public static void Main()
{
    (string firstname, string lastname, int height) = GetSomeData("Alex");
    Console.WriteLine($" {firstname} - {lastname} - {height}");
}

private static (string, string, int) GetSomeData(string name)
{
    if (name == "Alex")
        return ("Alexander", "Subbotin", 196);

    return (string.Empty, string.Empty, 0);
}
```

Properties

Специальный член для реализации инкапсуляции. MSDN

Состоит из двух accessor: `get`/`set`

```
public class Sample
{
    private int _x;

    public int PropertyX
    {
        get { return _x; }
        set { _x = value;}
    }
}

var sample = new Sample();
sample.PropertyX = 1;
```

```
public class Sample
{
    private string firstName;
    private string lastName;

    // AutoProperty
    public int PropertyX {get;set;}
    public int PropertyZ {get; private set;}

    // Expression bodied
    public DateTime Time => DateTime.UtcNow;
    public string Name => $"{firstName} {lastName}";

    // Через epression методы
    public int X
    {
        get => name;
        set => name = value;
    }
}
```

Все пользуются свойствами, никто не пишет свои get/set методы.
Но надо понимать, что при сравнении с просто полями:

- Свойства могут быть доступны только для чтения или только для записи, в то время как поля всегда доступны и для чтения, и для записи
- Свойство, являясь по сути методом, может выдавать исключения, а при обращениям к полям исключений не бывает
- Свойства нельзя передавать в метод в качестве параметров с ключевым словом `out` или `ref`

Рекомендации:

- Используйте всегда свойства для доступа к полям
- Используйте AutoProperty всегда, если их хватает
- НО! Не используйте property чаще, чем нужно !!!
    - Свойства - короткие и простые; длинные простыни кода или занимающие долгое время выполнения должны оформляться только методами!
    - Кидающий исключения код выносите в методы (кроме, наверно, проверки на `disposed`)
    - Напишите метод, если повторный вызов на тот же экземпляр может вернуть другое значение, например, `DateTime.Now` - это косяк, должен быть `DateTime.Now()`
- Не создавайте новых экземпляров объектов в свойствах

`readonly`

Поле класса, помеченное `readonly` может быть изменено **только** в конструкторе:

```
internal class SomeType
{
    private readonly int _value;

    public SomeType(Int x)
    {
        _value = x;
    }

    public void Set(int x)
    {
        _value = x; // Нельзя, компилятор будет ругаться
    }
}
```

const

- Константы задаются на момент компиляции.
- Могут использоваться только примитивные типы: `int`, `double`, `string`, etc.
- Должны быть здесь же инициализированы

```
internal class SomeType
{
    internal const int X = 10;
    const int months = 12, weeks = 52, days = 365;

    const double daysPerWeek = (double) days / (double) weeks;
}

SomeType.X // 10
```

static

static указывает что данный элемент относится не к конкретному экземляру, а к типу в целом. Поэтому обращие к статическим элементам / методам происходит без создания экземпляра.

static member

- Все экземпляры класса будут обращаться к единому статичному полю / методу
- Метод может использовать внутри себя только статические поля класса

```
public class Automobile
{
    public static int NumberOfWheels = 4;
    public static int SizeOfGasTank { get { return 15; } }
    public static void Drive() { }
    public static event EventType RunOutOfGas;
}
...
Automobile.Drive(); // Обращаемся через тип
int i = Automobile.NumberOfWheels;
```

`static` class

- В статическом классе можно объявлять только статические члены.
- Не может быть инстанциирован (нельзя использовать в качестве локальной переменной или параметра метода)
- Класс должен быть `sealed`
- Должен не реализовывать никаких интерфейсов
- нельзя сделать статическую стуктуру (всегда можно создать экземпляр)

Используется для написания хелперов с общей логикой без состояния.

```
public static class MyHelper
{
    public static string EncodeObject<T>(T value) {...}
}

string result = MyHelper.Encode(myValue);

double dub = -3.14;
Console.WriteLine(Math.Abs(dub));
```

`static` конструктор

Статический конструктор используется для инициализации статических полей класса

- Вызывается в **неопределенный** момент времени до использования. В clr реализовано, что он непосредственно вызывается перед первым использованием класса. Повлиять на это никак нельзя.
- Ему нельзя задавать модификатор доступа
- Ему нельзя передавать параметры
- Может быть только один конструктор на тип

```
public static class MyHelper
{
    public static readonly string Format;
    static MyHelper()
    {
        Format = Configuration.Format;
    }
    public static string EncodeObject<T>(T value) {...}
}
```

Надо иметь в виду:

- CLR гарантирует, что статический конструктор выполнится только один раз для всех потоков
- Если в таком конструкторе происходит исключение CLR считает весь тип непригодным и при попытке доступа к любым / полям методам будет кидать исключение
- Во время его вызова CLR накладывает исключительную блокировку на весь тип для всех остальных потоков в рамках домена приложения
- Поэтому возможны взаимные блокировки, нельзя писать код, который полагается на определенный порядок вызовов таких конструкторов

Рекомендации:

- Не используйте статические классы, кроме сценариев хелперов
- Не используйте статические классы для реализации singleton!
- Они ломают тестируемость и модульность приложения
- Они не ложатся в концепции di
- Они создают зависимости, которыми очень сложно управлять и неочевидно как отлаживать
- Не используйте статические конструкторы
- SOF discussion

partial

Частичные классы. `partial` позволяет создавать класс (структуру или интерфейс), расположенный в нескольких файлах, которые компилятор соединит в один.
Для удобства редактирования кода и автогенерации кода.

```
public parital MyClass
{
    public string MethodA() {}
}
```

```
public parital MyClass
{
    public string MethodB() {}
}
```

Наследование, полиморфизм

В C# Нет множественного наследования классов.
Есть ряд ключевых слов для управления связей между классами

- `virtual` - член может быть переопределен в производном типе
- `override` - переопределение члена в производном типе
- `abstract` - базовый класс, не предполагающий инстанцирование
- `sealed` - закрытый класс, от которого нельзя наследоваться
- `new` - метод/поле не связаны с членом базового класса
- Виртуальные методы медленнее невиртуальных (call / callvirt), целесообразно делать их как можно меньше

Базовый пример наследования:

```
internal class A
{
    internal A(int x)
    {
        X = x;
    }
    internal int X {get;set;}
}

internal class B:A
{
    internal B(int x, int y):base(x)
    {
        Y = y;
        // base.X;
    }
    internal int Y {get;set;}
}
```

Базовый пример полиморфизма:

```
public class A
{
    public virtual string Method => "this A";
}

internal class B:A
{
    public override string Method => "this B";
}

A valueAB = new B();
Console.WriteLine(valueAB.Method); // this B
A valueA = new A();
Console.WriteLine(valueA.Method); // this A
B valueB = new B();
Console.WriteLine(valueB.Method); // this B
```

Пример оператора `new`:

```
public class A
{
    public virtual string Method => "this A";
}

internal class B:A
{
    public new string Method => "this B";
}

A valueAB = new B();
Console.WriteLine(valueAB.Method); // this A
A valueA = new A();
Console.WriteLine(valueA.Method); // this A
B valueB = new B();
Console.WriteLine(valueB.Method); // this B
```

Общее про наследование:

- Не вызывайте виртуальные методы из конструктора
- Наследование - самая сильная связь между классами, используйте ее только там, где это реально нужно

`sealed`

- Если применяется на класс: запрет наследования от этого класса
- Если применяется на member: запрет на переопределение элемента в производных классах, используется только совместно с override
- Рихтер рекомендует делать все классы по-умолчанию `sealed` (на практике никто не заморачивается)

```
public class A
{
    public virtual string Method => "this A";
}
public class B:A
{
    public override sealed string Method => "this B";
}
public sealed C:B { }
```

abstract

Позволяет создать базовый незаконченный класс, который должен быть реализован в наследниках.

- Абстрактный класс не может быть инстанциирован
- Может содержать абстрактные методы, которые не содержат реализации (производный класс должен будет переопределить все такие методы)
- Может содержать базовые поля и реализации методов (эти члены можно не переопределять в производных классах)

```
public abstract class A
{
    public int X { get; set; }
    public abstract void DoWork(int i);

    public string MethodWithBasicBehaviour()
    {
        return "some string";
    }
}
```

- при этом переопределении абстрактного метода производный класс должен использовать `override`. Ключевого слова `virtual` нет, а поведение похожее.

```
public abstract class A
{
    public abstract void DoWork(int i);
}

public class B:A
{
    public override void DoWork(int i)
    {
        Console.WriteLine(i);
    }
}
```

Interface

Интерфейсы опередеяют некоторый функционал, не имеющий конкретной реализации.
В отличие от наследования связь между классами не `is-a`, а `can-do`

- Определяют некоторый контракт
- Есть множественная реализация интерфейсов
- У методов интерфейса нет модификаторов доступа
- Задается только сигнатура методов
- Могут содержать методы, свойства, события, индексаторы
- Класс реализующий интерфейс должен реализовать все его члены

Пример:

```
public interface IEquatable<T>
{
    bool Equals(T obj);
}

public class A: IEquatable<A>
{
    public int X {get;set;}

    // Очень плохая реализация для примера
    public bool Equals(A obj)
    {
        if (this.X == obj.X)
            return true;
        return false;
    }
}
```

Случай, когда в интерфейсах есть одинаковые методы:

```
interface IControl { void Paint(); }
interface ISurface { void Paint(); }

class SampleClass : IControl, ISurface
{
    public void Paint()
    {
        Console.WriteLine("Paint");
    }
}
...
SampleClass sc = new SampleClass();
IControl ctrl = (IControl)sc;
ISurface srfc = (ISurface)sc;
sc.Paint();
ctrl.Paint();
srfc.Paint();
```

- Можно переопределить отдельно для каждого интерфейса.
- Надо иметь в виду, что явное указание интерфейса при реализации обязывает указывать интерфейс при вызове экземплярного метода, поэтому всегда предпочтительна "неявная" реализация интерфейса

```
public class SampleClass : IControl, ISurface
{
    void IControl.Paint() => System.Console.WriteLine("IControl.Paint");
    void ISurface.Paint() => System.Console.WriteLine("ISurface.Paint");
}
...
SampleClass obj = new SampleClass();
//obj.Paint();  // Compiler error.
IControl c = (IControl)obj;
c.Paint();  // Calls IControl.Paint on SampleClass.
ISurface s = (ISurface)obj;
s.Paint(); // Calls ISurface.Paint on SampleClass.
```

Если разные члены с одним именем, то придется явно указывать интерфейсы:

```
interface ILeft
{
    int P { get;}
}
interface IRight
{
    int P();
}

class Middle : ILeft, IRight
{
    public int P() { return 0; }
    int ILeft.P { get { return 0; } }
}
```

Абстрактный класс VS Реализация интерфейса

- Абстрактные классы могут иметь поля и базовую реализацию методов
- В абстрактных классах можно задавать видимость элементов
- Базовый класс может повышать видимость при наследовании и в то время, как при реализации интерфейса должны оставить такой же видимости, как интерфейс (но вообще это довольно бесполезная возможность)
- При наследовании от абстрактного класса производный должен переопределить только абстрактные члены

SOF Discussion 1, 2, 3

Перегрузка операторов

Можно перегрузить стандартные операторы для своего класса.

- `public static` обязательно указываются
- Для операторов возвращающих обьект - Не надо изменять состояние передаваемых в параметры объектов - надо создать новый объект !!!
- Есть ряд операторов, которые нельзя переопределить `=`, `?:`, etc

```
public class Example
{
    public int X { get; set; }
    public static Example operator +(Example f, Example s)
    {
        return new Example { X = f.X + s.X };
    }
    public static bool operator >(Example f, Example s) => return (f.X > s.X);
    public static bool operator <(Example f, Example s) => return (f.X < s.X);
}
```

Использование в коде:

```
static void Main(string[] args)
{
    Example e1 = new Example { X = 4 };
    Example e2 = new Example { X = 7 };
    Example e3 = e1 + e2;
    Console.WriteLine(e3.X);  // 11

    bool result = e1 > e2;
    Console.WriteLine(result); // false
}
```

Еще примеры:

```
public static int operator +(Example e, int value)
{
    return e.X + value;
}

public static Example operator ++(Example e)
{
    return new Example { X = e.X + 10 };
}
```

Еще можно переопределить true / false:

```
public class Example
{
    public int X { get; set; }

    public static bool operator true(Example e) => return e.X > 0;
    public static bool operator false(Example e) => return e.X <= 0;
}

var value = new Example { X = 0 };
if (value)
    Console.WriteLine(true);
else
    Console.WriteLine(false);
```

Перегрузка преобразований типов

- Позволяет задавать `implicit` | `explicit` преобразования между типами
- должен быть `public static`

```
public static implicit|explicit operator TypeTo(BaseType value)
{
    return <TypeToObject>...;
}
```

```
public class Example
{
    public int X { get; set; }

    public static implicit operator Example(int value)
    {
        return new Example { X = value };
    }

    public static explicit operator int(Example value)
    {
        return value.X;
    }
}

Example value = new Example { X = 3 };

int intX = (int)value;
Example result = intX;
Console.WriteLine(result.X);  // 3
```

Extension methods

- Методы расширения позволяют добавлять методы в уже существующие типы
- Метод расширения может жить только в статическом классе и сам быть статическим

```
public static class StringHelper
{
    public static string Left(this string value, int size)
    {
        if (string.IsNullOrEmpty(value))
            return value;

        return value.Length <= size
               ? value
               : value.Substring(0, size);
    }
}

string s = "my string";
Console.WriteLine(s.Left(5)); // my st
```

Аттрибуты

MSDN Atributes

Есть много готовых атрибутов `[Serialized]`, `[Flags]` и др.

- Можно создавать кастомные атрибуты, наследуясь от `System.Attribute`
- Стандартное соглашение, что аттрибуты именуются словом Attribute, компилятор умеет убирать его
- Должны содержать хоть один конструктор

```
[System.Serializable]
public class SampleClass
{
    // Objects of this type can be serialized.
}

public class FlagsAttribute : System.Attribute
{
    public FlagsAttribute() {}
}
```

- Обычно атрибут применяется на элемент после него, но можно явно задать, на что именно будет применяться атрибут: `assembly`, `module`, `field`, `event`, `method`, `param`, `property`, `return`, `type`

```
// applies to method
[method: SomeAttr]
int Method2() { return 0; }

// applies to return value
[return: SomeAttr]
int Method3() { return 0; }
```

- `[AttributeUsage]` указывает область применимости атрибута для компилятора
- Если не пометить, то атрибут можно будет применять к любому объекту
- `Inherited` - должен ли атрибут применятся к производному классу (по-умолчанию true) при overrid'инге методов
- `AllowMultiple` - можно ли навешать несколько одинаковых атрибутов на член (по-умолчанию false)

```
namespace System
{
    [AttributeUsage(AttributeTargets.Enum, Inherited = false)]
    public class FlagsAttribute : System.Attribute
    {
        public FlagsAttribute() {}
    }
}
```

`Inherited` example:

```
[AttributeUsage(AttributeTargets.Class | AttributeTargets.Method, Inherited=true)]
internal class TastyAttribute : Attribute {}

[Tasty]
[Serializable]
internal class BaseType
{
    [Tasty]
    protected virtual void DoSomething() { }
}

internal class DerivedType : BaseType
{
    protected override void DoSomething() { }
}    // И DerivedType и DoSomething будут помечены Tasty
```

- Можно задавать конструктор и публичные нестатические поля / свойства для атрибута
- В конструкторе можно использовать только маленький набор типов: `bool`, `char`, `byte`, SByte, `short`, UInt16, `int`, UInt32, `long`, UInt64, `float`, `double`, `string`, `Type`, `object`, `enum`
- Можно передавать в конструктор только константы и `typeof()` для получения типа
- При компиляции вызовется конструктор атрибута, и его сериализованный объект запишется в метаданные
- Можно массив константных типов передавать в конструктор, но лучше так не делать (!)

```
internal enum Color { Red }

[AttributeUsage(AttributeTargets.All)]
internal sealed class SomeAttribute : Attribute
{
    public SomeAttribute(String name, Object o, Type[] types)
    {
    }
}

[Some("Jeff", Color.Red, new Type[] { typeof(Math), typeof(Console) })]
internal sealed class SomeType {}
```

Как их использовать? `System.Attribute`

- `IsDefined`
- `GetCustmoAttributes`
- `GetCustmoAttribute`

```
public static String Format(Type enumType, Object value, String format)
{
    // Is [FlagsAttribute] applied to instance?
    if (enumType.IsDefined(typeof(FlagsAttribute), false))
    {
        // Yes; execute code treating value as a bit flag enumerated type.
    } else
    {
        // No; execute code treating value as a normal enumerated type.
    }
}

System.Attribute[] values = System.Attribute.GetCustomAttributes(typeof(myType));
```

Некоторые примеры использования:

- Маппинг объектов в БД
- Вызов unmanaged кода с помощью DllImportAttribute
- Настройки сериализации, какие поля и как сериализовать в xml, json
- Описание требований безопасности к методам / классам (используется в asp.net mvc)